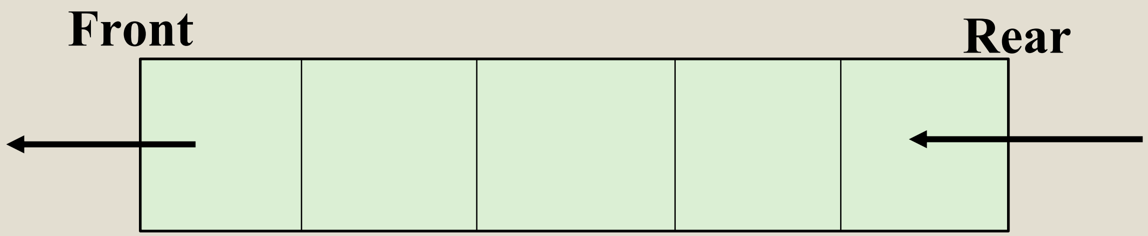
QUEUE & ITS APPLICATION

Queue

- It's a linear as well as non-primitive data structure.
- It's an ordered collection of items that works upon a simple formula called as **FIFO** (First In First Out).
- There are 2 ends: front & rear.
- Elements are inserted at **rear** and deleted from **front**.
- Queue can be a homogeneous/non-homogeneous, static/dynamic data structure.
- Queue can be created using Array and Linked List.
- Queue has 3 operations:
  - EnQueue (insert): check for queue full/overflow
  - DeQueue (delete): check for queue empty/underflow
  - Display/Peek
- Variants of Queue:

  1. Linear Queue 2. Circular Queue

  3. Priority Queue 4. Double-ended Queue



Queue

Stack

- LIFO (Last In First Out).
- There is only 1 end: top.
- Elements are inserted and deleted from top.
- Stack has 3 operations:
  - Push (insert): check for stack full/overflow
  - Pop (delete): check for stack empty/underflow
  - Display/Peek
- **Applications of Stack**
  - Parsing in a compiler
  - Java virtual machine (JVM)
  - Back button in a Web browser
  - Implementing function calls in a compiler

Queue

- FIFO (First In First Out).
- There are 2 ends: front & rear.
- Elements are inserted at rear and deleted from front.
- Queue has 3 operations:
  - Enqueue (insert): check for queue full/overflow
  - Dequeue (delete): check for queue empty/underflow
  - Display/Peek
- **Applications of Queue**
  - Data Buffers
  - Asynchronous data transfer (file IO, pipes, sockets)
  - Allotting requests on a shared resource (printer, processor)
  - Traffic analysis
  - Determine the number of cashiers to have at a supermarket

Linear Queue using Array

Queue declaration: public static final int MAXSIZE = 4;

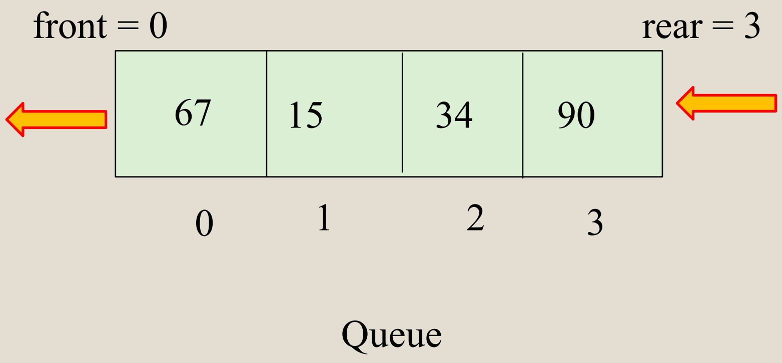
int queue [ ] = new int [MAXSIZE]; // Queue creation

Queue empty: public static int front = -1; // front & rear declaration

public static int rear = -1;

queue empty

Queue full: if (rear = = MAXSIZE - 1)




```
void display(int queue[ ])
{
   System.out.println ("Elements present in queue.");
   for (int i = front; i <= rear; i++)
         System.out.println (queue[i]);
}
```

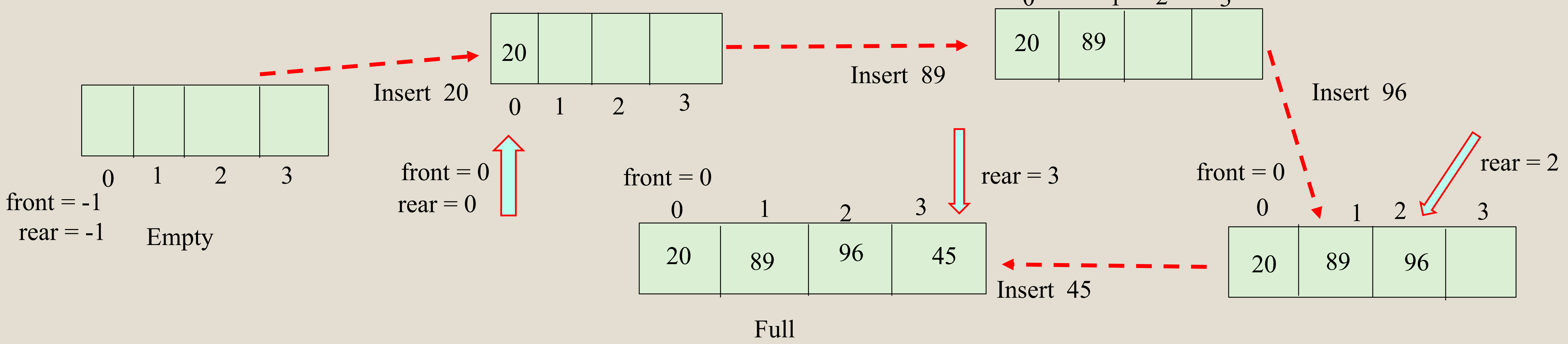
Linear Queue using Array

```
void insert (int queue[ ])
{
        if (isFull ())
            System.out.println (“Queue is Full!");
        else
        {
            Scanner sc = new Scanner (System.in);
            System.out.println ("Enter element");
            rear + +;
            queue [rear] = sc.nextInt();
        }

        if (rear = = 0)    front = 0;
}
```

```
boolean isFull ()
{
    if (rear = = MAXSIZE - 1)
        return true;
    else
        return false;
}
```

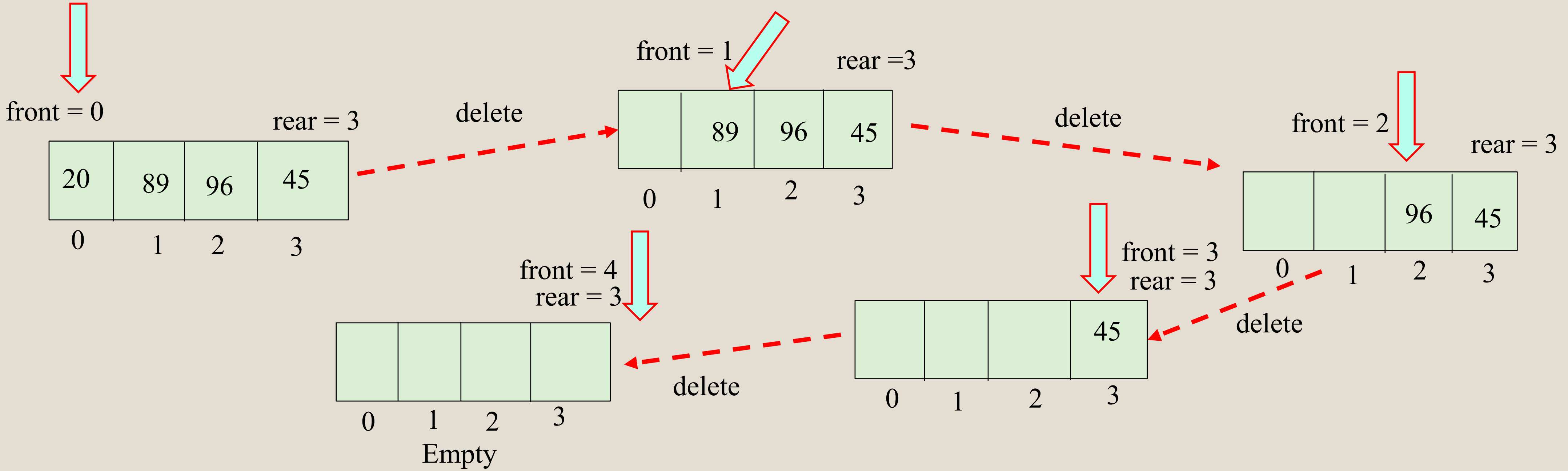
Note: If the Queue contains a single element, then both front & rear is at 0.



Linear Queue using Array

```
void delete (int queue[ ])
{
    if (isEmpty ())
        System.out.println ("Underflow!");
    else
    {
        System.out.println ("Deleted Element is "+ queue[front]);
        front + +;
    }
}
```

```
boolean isEmpty ()
{
    if (front = = -1 || front > rear || rear = = -1)
            return true;
    else
            return false;
}
```



Linear Queue using Linked List

Node declaration:

```
class node
{
    int info;
    node next;
}
```

Queue empty: node front = null; // front declaration

No Queue full condition is there except when node creating heap area is full

Node Insertion: just like inserting a new node at the end of the linked list

Node Deletion: just like deleting a node from the beginning of the linked list

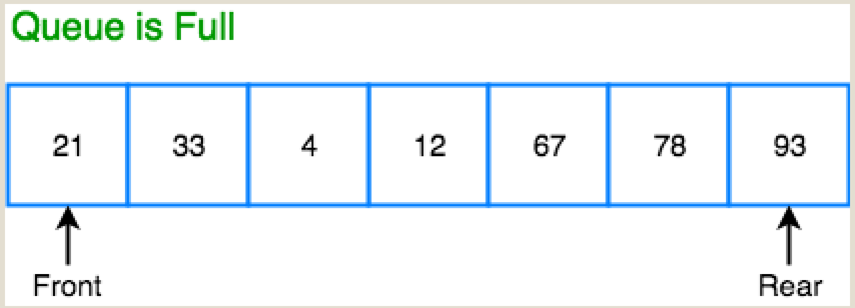
Linked Queue using Linked List

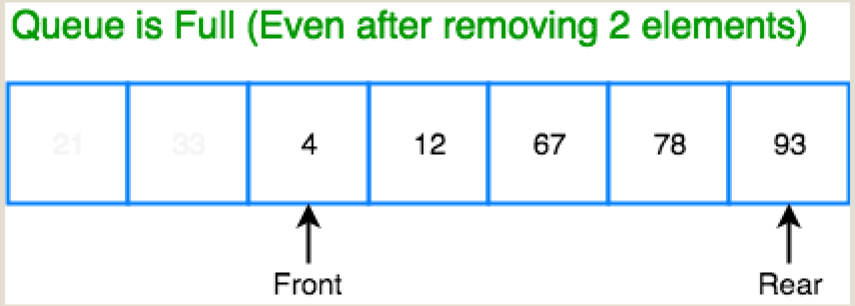
```
void delete (node front)
{
    if (front = = null)
        System.out.println (“List Empty");
    else
    {
        System.out.println ("Deleted Element is "+ front);
        front = front.next;
    }
}
void display (node front)
{
        node s = front;
        while (s != null)
        {
          System.out.println (s.info);
          s = s.next;
    }
}
```

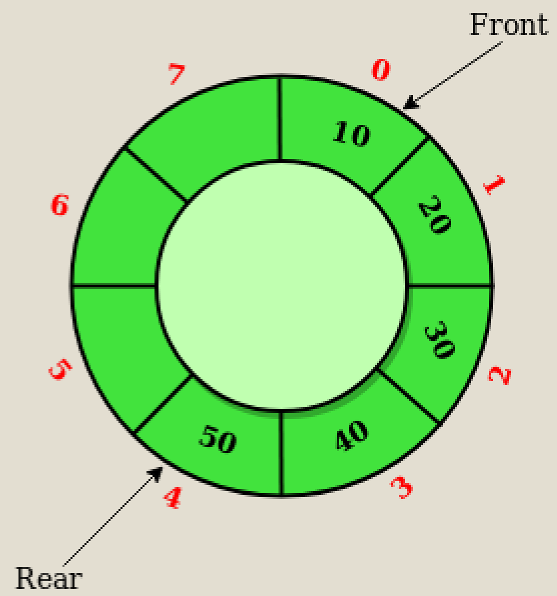
```
void insert (node front)
{
        Scanner sc = new Scanner(System.in);
        node temp = new node();
        node s = new node();
        temp.info = sc.nextInt();
        temp.next = null;
        if (front = = null)
        {
                front = temp;
        }
        else
        {
                s = front;
                while (s.next != null)
                {
                        s = s.next;
                }
                s.next = temp;
        }
}
```

Linear Queue vs Circular Queue

- In a Linear queue, once the queue is completely full, it's not possible to insert more elements.
- Even if we delete some elements, no new elements can be inserted.
- Because, we are moving the **front** of the queue forward and we cannot insert new elements, because the **rear** pointer is still at the end of the queue.
- So, the Circular Queue is used to overcome this issue, which also uses **FIFO** (First In First Out).
- The last position is connected back to the first position to make a circle.
- Application of Circular Queue:
  - Computer controlled **Traffic Signal System** uses circular queue
  - CPU scheduling and Memory management







Circular Queue

Queue declaration: public static final int MAXSIZE = 3;

```
int circ_queue [ ] = new int [MAXSIZE];        // Queue creation
```

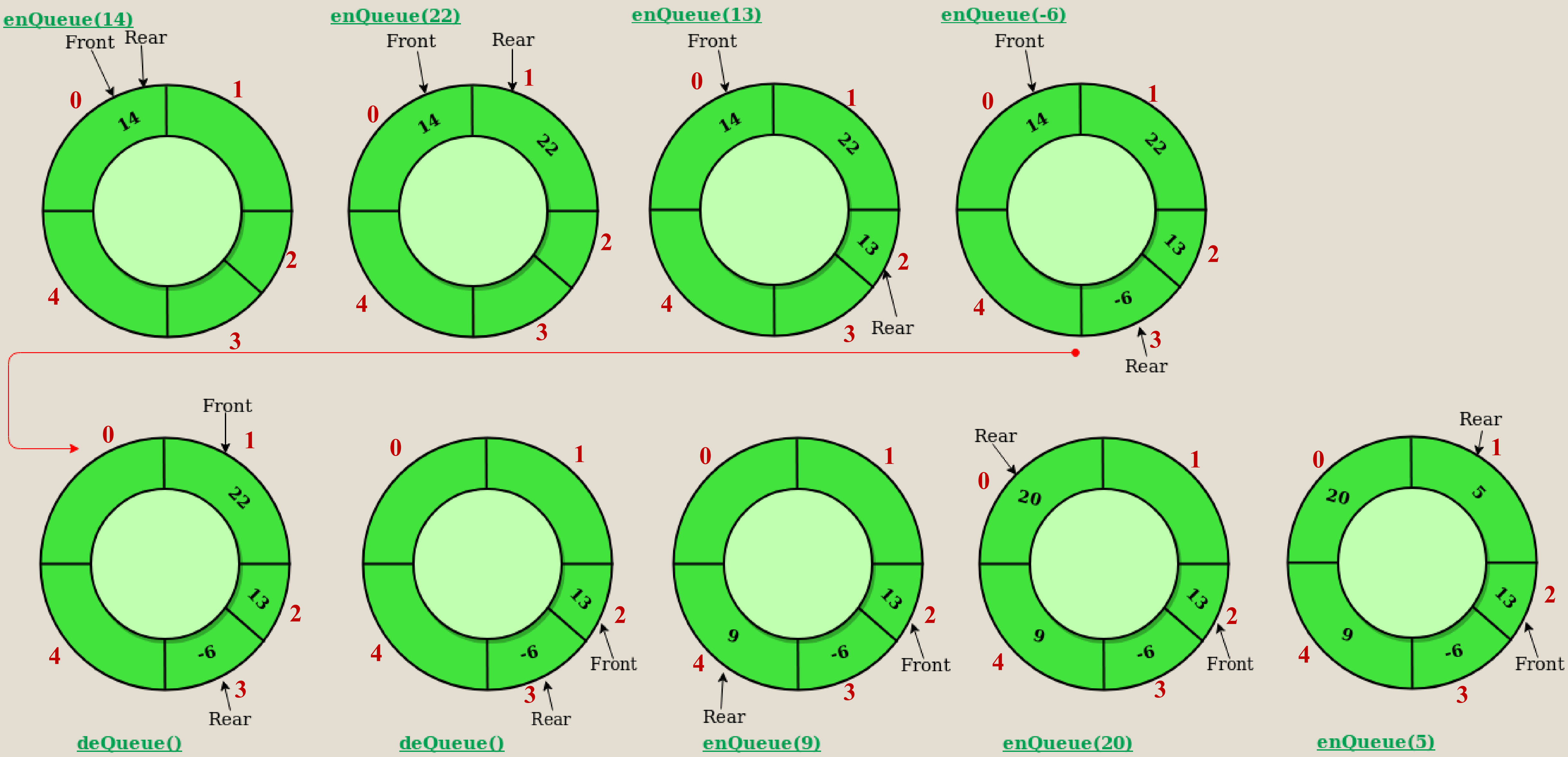
Queue empty:

```
public static int front = -1;                  // front & rear declaration
public static int rear = -1;
```

circular queue empty

- While inserting (enqueuing), we circularly increase the value of REAR index and place the new element in the position pointed to by REAR.
- While deleting (dequeuing), we return the value pointed by FRONT and circularly increase the FRONT index.
- Before enqueuing, we check if the queue is already full.
- Before dequeuing, we check if the queue is already empty.
- When enqueuing the first element, we set the value of FRONT to 0.
- When dequeuing the last element, we reset the values of both FRONT and REAR to -1.

Circular Queue



Circular Queue

```
public static void display (int circ_queue[])
{
    int i;
    if (isEmpty())
            System.out.println ("Empty Circular Queue");
     else
     {
            System.out.println ("Items are ");
            for (i = front; i != rear; i = (i + 1) % MAXSIZE)
                    System.out.println (circ_queue [i] + " ");
            System.out.println (circ_queue [i]);
     }
}
```

```
public static void insert (int circ_queue[])
        {
            if (isFull())
                    System.out.println (“Circular Queue is Full!");
            else
            {
                    if(front = = -1)
                            front = 0;
                    Scanner sc = new Scanner(System.in);
                    System.out.println ("Insert element");
                    rear = (rear + 1) % MAXSIZE;
                    queue [rear] = sc.nextInt();
            }
        }
        public static boolean isFull()
        {
            if ((front = = 0 && rear = = MAXSIZE - 1) || (front = = rear + 1))
                return true;
           else
                return false;
        }
```

Circular Queue

```
public static void delete (int circ_queue[])
{
        if (isEmpty())
          System.out.println (“Circular Queue is Empty!");
        else
        {
          System.out.println ("Deleted "+ circ_queue [front]);
          if (front = = rear)
          {
             front =  - 1;
             rear =  - 1;
          }
          else
             front = (front + 1) % MAXSIZE;
        }
}
```

```
public static boolean isEmpty()
    {
        if (front = = -1)
            return true;
        else
            return false;
    }
```